Using Versions in Update Transactions

François Llirbat, Eric Simon, Dimitri Tombroff

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Using Versions in Update Transactions

Francois Llirbat, Eric Simon, Dimitri Tombroff

N° 2940

Juillet 1996

PROGRAMME 1

ISSN 0249-6399

Using Versions in Update Transactions

Francois Llirbat*, Eric Simon*, Dimitri Tombroff*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Rodin

Rapport de recherche n°2940 — Juillet 1996 — 41 pages

Abstract: This paper proposes an extension of the multiversion two phase locking protocol, called EMV2PL, which enables update transactions to use versions while guaranteeing the serializability of all transactions. The use of the protocol is restricted to transactions, called *write-then-read* transactions that consist of two consecutive parts: a write part containing both read and write operations in some arbitrary order, and an abusively called read part, containing read operations or write operations on data items already locked in the write part of the transaction. With EMV2PL, read operations in the read part use versions and read locks acquired in the write part can be released just before entering the read part. We prove the correctness of our protocol, and show that its implementation requires very few changes to classical implementations of MV2PL. After presenting various methods used by application developers to implement integrity checking, we show how EMV2PL can be effectively used to optimize the processing of update transactions that perform integrity checks. Finally, performance studies show the benefits of our protocol compared to a (strict) two phase locking protocol.

Key-words: Transactions, concurrency control, integrity checking, performance analysis.

(Résumé : tsvp)

*{Francois.Llirbat}{Eric.Simon}{Dimitri.Tombroff}@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Utilisation de versions par des transactions d'écriture

Résumé : Cet article propose un protocole de contrôle de concurrence multiversion appelé EMV2PL. Ce protocole permet d'exécuter efficacement des transactions d'écriture particulières composées de deux parties: une partie d'écriture qui contient des opérations d'écriture et de lecture dans un ordre arbitraire et une partie de lecture qui contient des opérations de lecture et/ou des opérations d'écriture sur des objets déjà verrouillés dans la partie d'écriture de la transaction. Avec EMV2PL, ces transactions (1) relâchent leurs verrous de lecture avant d'exécuter leur partie de lecture et (2) exécutent les opérations de lecture de la partie de lecture sur des versions, sans prendre de verrous. Nous montrons la correction de ce protocole et le fait que sa mise en oeuvre ne nécessite que de légères modifications des algorithmes et des structures de données utilisées par le protocole classique MV2PL. Nous présentons différentes méthodes employées pour vérifier l'intégrité des données dans les applications de bases de données et montrons comment EMV2PL optimise ces applications. Enfin, une étude par simulation nous permet d'illustrer les gains en performances de notre protocole par rapport à un protocole de verrouillage à deux phases.

Mots-clé : Transactions, contrôle de concurrence, contrainte d'integrité, étude de performance.

1 Introduction

Multiversion concurrency control schemes exploit the use of versions to increase concurrency between transactions [BHG87]. The most popular scheme is the *Multiversion two phase commit* protocol[1] (MV2PL), which eliminates blockings between *read-only* transactions (i.e., transactions that do not change the state of the database) and update transactions. In this protocol, read-only transactions access old versions of data items whereas update transactions read and write the most recent versions. Because read-only transactions access a past state, they never conflict with update transactions and they do not need to take locks. As a result, they are never blocked, nor do they block concurrent update transactions. In contrast, update transactions may conflict with each other and obey the strict two phase locking policy (henceforth, S2PL). Several performance analysis have shown the value of MV2PL over the standard S2PL protocol ([CM86], [BC92a]). Acknowledging its virtues, several database vendors have incorporated different variants of MV2PL into their products[2]. However, the benefits of multiversion concurrency control for concurrent update transactions is still controversial [BBG+95]. Thus, no commercial DBMS implements a scheme in which update transactions read old versions while guaranteeing serializable executions[3].

In this paper, our main contribution is to propose an extension of the MV2PL protocol, called EMV2PL, which enables update transactions to use versions while guaranteeing the serializability of all transactions. However, the use of the protocol is restricted to a particular class of update transactions called *write-then-read* transactions (noted W|R transactions). A transaction in this class consists of two consecutive parts: the first part (called the write part) contains both read and write operations in some arbitrary order, and the second part (abusively called the read part) only contains read operations or write operations on data items already locked in the first part of the transaction. Thus, no new write lock is acquired in the second part of the transaction. In our protocol, W|R transactions take advantadge of versions in two ways: (i) they release their read locks before executing their read part,

[1] In [BHG87] it is called multiversion mixed method; multiversion two phase locking is reserved for a different protocol.

[2] E.g., SQL-92 Rdb ([HE91]), Postgres and Illustra ([Ill94]), Oracle when used with SET TRANSACTION READ ONLY ([Par89])

[3] Several products (e.g., Borland's InterBase 4 ([Tha94]), Microsoft's Exchange System) offer the so-called "Snapshot Isolation" in which update transactions read old versions and write new versions. Oracle provides a similar technique called "READ CONSISTENCY". However, Snapshot Isolation may produce non serializable executions as shown in [BBG+95].

and (ii) they execute their read part without taking any lock, as read-only transactions do with MV2PL. Using a careful performance analysis, we demonstrate that the combination of both effects results in a significant increase in concurrency, and reduces the probability of deadlocks, as compared to the S2PL protocol. A notable feature of our protocol is its simplicity, as attest the few modifications to a classical MV2PL implementation that are required to implement it. We consider it as a virtue since our intention in this research is not to invent yet another new concurrency control protocol but rather to enhance existing implementations to better match application needs.

Write-then-read transactions are common in many database applications. Update transactions that perform calculations at the end is one typical case. For instance a transaction that performs financial commitments (i.e., database updates) may then return statistics values to the user such as sum of commitments or remaining budget. Another interesting case is applications in the area of data mining using sophisticated statistical methods. There, statistical tests are triggered as soon as some relevant situation is detected in the database as a result of updates. Additionally, one may specify when to inform decision-makers about actual findings, e.g., when the value of some statistical test exceeds a threshold. The triggering of statistical tests as well as the alerting of decision-makers can be appropriately implemented by active rules triggered at the end of update transactions. Since the triggers either perform statistical computations or alert users, those update transactions are write-then-read transactions.

A second important contribution of this paper is to show how EMV2PL can be effectively used to optimize an application's transaction throughput when update transactions verify integrity constraints. Although EMV2PL has a wider applicability, we consider this problem a valuable illustration of its potential, because an important part of the database applications is devoted to the enforcement of integrity constraints that guarantee that each transaction is consistent. We present various methods, procedural and declarative, used by application developers, to implement integrity constraints. We then consider integrity constraints, which can possibly trigger a transaction rollback if a constraint is violated, and whose enforcement only requires database read operations. Our performance measurements show that, if the probability that a transaction violates a constraint is small, checking integrity at the end of transactions that run under EMV2PL generally achieves a higher total transaction throughput than all other methods. Otherwise, checking integrity right at the beginning of transactions, and using S2PL, is usually better. This result is important for three reasons. First, integrity can always – and sometimes has to – be checked at the end of transactions. Second, it can easily be implemented using declarative deferred assertions and deferred triggers, thereby benefiting from the advantages of the declarative implementation of integrity. On the contrary, integrity constraints cannot always be checked at the beginning of transactions and when it can, this cannot be implemented neither with assertions nor triggers. Last, EMV2PL makes deferred assertions and triggers quite attractive from a transaction throughput performance perspective.

The remainder of this paper is structured as follows. In section 2, we present a motivating example and give an overview of our method. In section 3, we formally define our protocol, prove its correctness, and show that it can be implemented as a slight extension of multiversion two phase locking. In section 4, we discuss how EMV2PL can be applied to integrity checking. Section 5 presents our performance evaluation. Section 6 relates our work to other work, and we conclude in Section 7.

2 Problem Statement and Solution Overview

We present a motivating example in the framework of an information system representing the activity of an industry which must manage, sell, and distribute a product worldwide. We assume that the industry holds a set of widely distributed stores. We focus here on the processing of entry orders. The following relations are used.

```
Order (orderkey, custkey, orderdate, ...)
Lineitem (orderkey, itemkey, linenumber, qty, ...)
Totals_item (itemkey, total_qty, ...)
Totals_cust (custkey, total_balance, authorized_limit, ...)
```

Orders are registered in the database using two relations *Order* and *Lineitem*[4]. We suppose that the system records summary information about items and customers in two particular relations. A first relation, *Totals_item*, records totals for each item such as the total quantity available in stock over all the stores[5]. A second relation, *Totals_cust*, records totals and limits for each customer, e.g., the total balance for items that have been delivered[6], and the minimal (negative) limit accepted for the balance of that customer. The total balance in *Totals_cust* is decremented when the receipt date is notified in *Lineitem*, and incremented when the payment is received. The total quantity in *Totals_item* is updated according to the flow of items (in and out) in each individual store. We assume that the contention is high on the two tables *Totals_item* and *Totals_cust*.

A transaction program, *entry_order*, inserts tuples into relations *Order* and *Lineitem*. Then, the transaction performs two checks. First, for each insertion in *Lineitem*, it compares the quantity ordered with the total quantity available for that item in relation *Totals_item*. Depending on the value of total_qty, a shipping date is estimated or the order for that item is refused. Then, the transaction checks the total balance and the authorized limit for the customer in *Totals_cust* and compares it with the total amount on order in the transaction. If the check is not successful, the transaction is aborted. We suppose that the transaction performs insertions first and then checks.

Suppose that transactions run in isolation degree 3 and obey the strict two phase locking policy [BHG87]. Whenever *entry_order* executes, checks are performed,

[4]The schema for these two relations roughly follows the indications of [TPC95].

[5]There might be one such relation per geographical area such as Asia, Europe, etc

[6]A customer can be allowed to pay within an interval of time following the date at which the ordered items have been delivered.

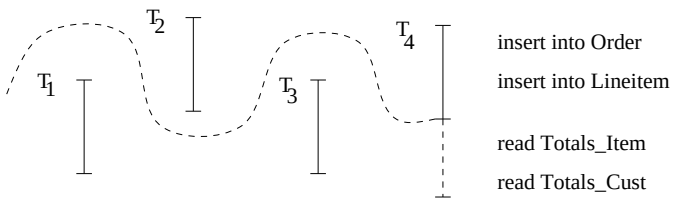


Figure 1: T_4 is serialized after T_2 and before T_1 and T_3.

which amounts to read one item in $Totals\_cust$ and several items in $Totals\_item$. Thus, executions of $entry\_order$ may be conflicting with concurrent executions of update transactions on both $Totals\_item$ and $Totals\_cust$ since they respectively want to read and write these relations. When a conflict occurs between two transactions, one transaction is blocked and waits that the other one releases its locks (when committing or aborting). Thus, running instances of $entry\_order$ augment the lock contention on $Totals\_item$ and $Totals\_cust$ and impede the transactional traffic in the database system.

In this example, let us observe that $entry\_order$ is a W|R transaction. Our basic optimization idea allows the read part of such transactions to read versions of database items exactly as read-only transactions do in the MV2PL protocol. In this protocol, transactions are either *read-only* transactions or *update* transactions. Each read-only transaction T is assigned a startup timestamp $sn(T)$ when it begins to execute, and each update transaction T is assigned a timestamp $tn(T)$ when it commits. Update transactions are serialized together through the usual strict two phase locking protocol. Instead of simply updating the data items, update transactions create new *versions* of the data items. These versions are timestamped with the timestamp tn of their creator. Hence, several ordered versions may exist for a single data item. When a read only transaction wishes to read an item, it simply reads the most recent version having a timestamp smaller than its startup timestamp. Thus, a read-only transaction reads only versions created by update transactions that committed before it started. The major advantage of this protocol is that read-only transactions do not acquire locks.

Coming back to our example, suppose that three update transactions on $Totals\_item$, noted T_1, T_2, and T_3, execute concurrently with an instance of $entry\_order$, noted T_4. The concurrent execution of these transactions is shown on Figure 1. Let us apply the principle of MV2PL to the read part of T_4. Relation $Totals\_item$ is not locked by T_4 since T_4 will read a version of $Totals\_item$ that corresponds to the snapshot represented by the dotted line in Figure 1. Thus, the possible conflicts on $Totals\_item$ may only occur between T_1, T_2, and T_3, and T_4 is not capable of blocking these transactions. Furthermore, the above execution is correct as far as serialization is concerned[7]. Transaction T_2, which commits before the read part of

[7]Note that *Snapshot Isolation* would not guarantee serializability on this example. With *Snapshot Isolation*, each query uses its start time as timestamp. Suppose for example that T_3 modify

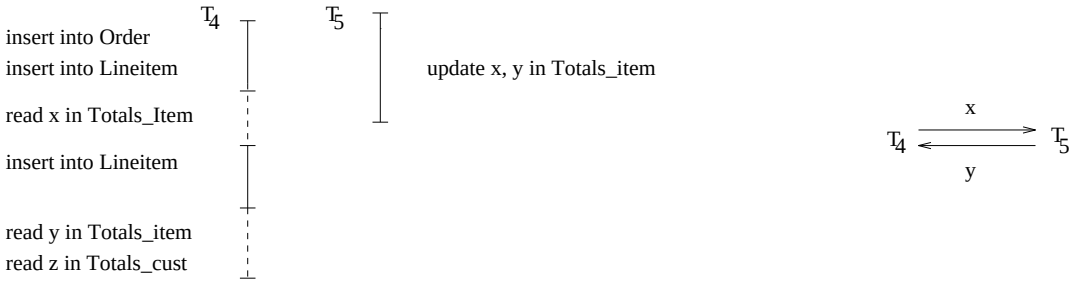


Figure 2: T_4 does not see T_5's modification on y but sees T_5's modification on z.

T_4 starts is serialized before T_4. Transactions T_1 and T_3, which commit after the read part of T_4 starts are serialized after T_4 because they commit after T_4 acquired all its locks.

However, the usage of MV2PL to monitor the read parts of W|R transactions can raise some problems. First, note the importance of our assumption that only new read locks are acquired. Suppose that T_4 is changed as follows. The first check on *Totals_item* is performed after each insertion of a tuple into *Lineitem* and the second check is performed at the end of the transaction. Then, the write parts and the read parts of the transaction are interleaved. This means that a new write-lock could be acquired after the first read on *Totals_item*. The problem is that a serialization fault might occur if a concurrent transaction, say T_5, executes and updates both *Lineitem* and *Totals_item*, as shown on figure 2. The edges of the serialization graph[8], labelled with the item that causes a conflict, are shown in this figure. Thus, although the semantics of the transaction is unchanged by the new implementation of T_4, our optimization opportunity is lost.

Even if all the checks are performed at the end of the transaction, the main problem may still occur when concurrent write-then-read transactions are executed. Suppose we have two concurrent transactions T_1 and T_2 as depicted on figure 3. T_1 reads a version of x created before T_2 since T_2 commits after the read part of T_1 starts. Using the same argument, T_2 reads a version of y created before T_1 started. Thus there is a cycle in the serialization graph and the execution is not correct.

In this example, the problem comes from the read(x) operation issued by the read part of transaction T_1. This operation makes T_1 serialized before T_2, although they started their read parts in the opposite order. The basic idea of our protocol is to dynamically prevent such "critical" reads from happening.

The *Snapshot Isolation* (SI) technique would also produce a serialization fault on the example of Figure 3. With SI, all reads of a transaction T read the most recent

both *Total_item* and *Totals_cust*. If T_3 commits after "read Total_item" is started but before "read Totals_cust" is started, "read Totals_cust" will not see T_3 changes while "read Totals_item" could see them, resulting in a serialisation fault

[8]This graph representing the serialization order of transactions is defined in Section 3

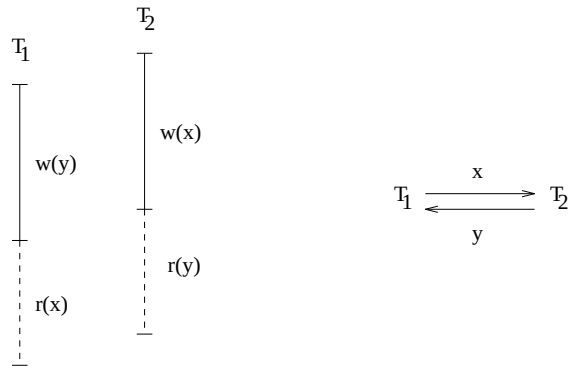


Figure 3: Concurrent execution of Write-then-Read Transactions.

versions created before the start time of T. Updates in T which create new versions timestamped with T's commit time. However, T is allowed to commit only if no other transaction whose commit timestamp belongs to the interval [T − start_time, T − commit_time] wrote a data item also written by T. Otherwise, T is aborted. Suppose that SI is used on the example of Figure 3: the read of x (resp. of y) returns the most recent version created before T_1 (resp. T_2) thus resulting in a serialization fault. In fact, as noted in [BBG+95], SI allows an important concurrency anomaly called *constraint violation*. Suppose there is a constraint between x and y (e.g., $x < y$) then after the execution of T_1 and T_2, this constraint might be violated.

Coming back to our protocol, we have so far sketched out how the conflicts between the read part of W|R transactions and other transactions can be eliminated using versions. However, there is a second advantage of using versions. Intuitively, the effect of our protocol is to make the read part of a W|R transaction access a snapshot of the database as of the time the W|R transaction reaches the end of its write part. From that point in time and until it commits (or aborts), the W|R transaction will ignore the database changes from concurrent transactions. Thus, the W|R transaction may safely release all its read locks *before* starting to execute the read part. Releasing locks earlier allows other transactions to modify the corresponding data items without having to wait for the W|R transaction to commit or abort. In contrast, a W|R transaction keeps all its write locks until it commits or aborts, as with S2PL.

3 Extended Multiversion Two Phase Lock Protocol

In this section we formally define the EMV2PL protocol and prove its correctness, we explain its behavior with respect to deadlocks and external consistency, and finally we briefly explain how it can be implemented.

3.1 Preliminaries

We first introduce basic notations and definitions, (see [BHG87] and [AS89]).

A database is a set of objects. A Transaction T_i is an ordered pair $(\Sigma_i, <_i)$ where Σ_i is the set of operations in T_i and $<_i$ is the execution order of these operations. Read or write operations executed by T_i are noted $r_i[x]$ or $w_i[x]$, respectively. Transactions terminate either by a commit (c_i for T_i) or an abort (a_i for T_i). Transactions are characterized as follows. A read-only transaction (noted R transaction) contains only read operations. An update transaction (noted W transaction) contains both read and write operations. Finally a *Write-then-Read* transaction is a particular W transaction that terminates by a part in which no new object is written. Formally a W|R transaction is a pair $(\Sigma_i, <_i)$ where $\Sigma_i = \Sigma_i^1 \cup \Sigma_i^2$, $o_1 <_i o_2$ for any operation $o_1 \in \Sigma_i^1$ and $o_2 \in \Sigma_i^2$, and $w_i[x] \in \Sigma_i^2$ only if $w_i[x] \in \Sigma_i^1$. The first part of a W|R transaction is called its *write* part and the second part its *read* part.

Let $T = \{T_1, T_2, ...T_n\}$ be a set of transactions. An *history* H of T is a partial order $(\Sigma, <_H)$ where Σ is the set of operations executed by transactions in T, and $<_H$ is the execution order of those operations. Two histories H_1 and H_2 are *conflict equivalent* if (i) they are defined over the same set of transactions and (ii) if o_1 and o_2 are two conflicting operations and $o_1 <_{H_1} o_2$ then $o_1 <_{H_2} o_2$. An history H_s is *serial* if for every two transactions T_i and T_j either all operations of T_i precede all operations of T_j or vice-versa. An history H is *serializable* if it is equivalent to a serial history. One determines if H is serializable by analyzing the *Serialization Graph* of H , noted SG(H). This is a directed graph where nodes are the committed transactions in H, and there is an edge $T_i \rightarrow T_j$ if one of T_i's operation precedes and conflicts with one of T_j's operation. An history H is serializable if its serialization graph is acyclic.

In a multiversion database, each write operation of an object x produces a new *version* of x. Thus for each object x, there is a list of versions written $x_i, x_j, ..$ where the subscript is the index of the transaction that wrote the version (thus, x_i represents the version of x created by T_i). A *multiversion history* (MV) is the sequence of operations on the versions of objects submitted by transactions. Each write operation $w_i[x]$ is mapped into $w_i[x_i]$ and each operation $r_i[x]$ is mapped into $r_i[x_k]$, for some k. A transaction T_j *reads-x-from* T_i in H if $r_j[x_i] \in H$. Notice the only conflicting operations in H are $w_i[x_i]$ and $r_j[x_i]$ for some x, T_i and T_j. Two MV histories are equivalent if they have the same operations. An MV history is *one-copy serializable* if it is equivalent to a serial history over the same set of transactions executed over a single version database. The *multiversion serialization graph* of an MV history H (MVSG(H)) is a directed graph whose nodes represent committed transactions. There is an edge $T_i \rightarrow T_j$ in MVSG(H) if

(i) $r_j[x_i] \in H$ for some x (that is, T_j reads-x-from T_i)

Such an edge is called an SG(H) edge in [BHG87]. Additional edges are defined as follows. For each object x, there is a total order (noted $\ll_x$) on all transactions that write x. One adds the edge $T_i \rightarrow T_j$ to MVSG(H) iff one of the following holds:

Operation Invocation	Operation Execution
$begin(T)$	get $sn(T)$ from TM
...	
$read(x)$	return x's version with largest version number $\leq sn(T)$
...	
$end(T)$	ϕ

Figure 4: Execution of an R transaction

(ii) for some x and T_k, T_i reads-x-from T_k and $x_k \ll_x x_j$
(iii) for some x and T_k, T_k reads-x-from T_j and $x_i \ll_x x_j$

These additional edges are called *version order edges*. An *MV* history is one-copy serializable if its multiversion serialization graph is acyclic ([BHG87]).

3.2 The EMV2PL Protocol

We now present our Extended MV2PL protocol, called EMV2PL. Figures 4 and 5 respectively show how operations issued by R and W transactions are processed by the Transaction Manager (TM). An R transaction first obtains a *start number* noted sn from TM. Then every read(x) gets the most recent version of x having a timestamp less than or equal to sn. Reads in a W transaction follow the usual S2PL protocol, whereas a write(x) creates a new version of x (if x is written for the first time). Before committing, a W transaction obtains its *transaction number* (noted tn), associates this number to each of its versions, and releases all its locks.

Figure 6 shows how the operations issued by a W|R transactions are processed. The write part of the transaction is processed as a W transaction. When the end of the write part is reached, the transaction signals it reached a *lockpoint*[9] to the TM and receives a transaction number tn. The transaction then releases all the S locks it has acquired so far. After that point, read and write operations are processed as follows. A read(x) operation invokes a function $check\_read(x)$ that checks if there is an uncommitted version[10] of x created by another transaction whose number is smaller than the caller's tn. In that case, *check_read* waits until that transaction commits. After that, the W|R transaction reads the most recent version of x with timestamp smaller than or equal to tn. A write(x) operation only modifies a version already created in the write part. Before committing, a W|R transaction associates its tn to each version it created and releases all its locks.

To maintain the tn's, the TM uses a monotonically increasing counter. Since W and W|R transactions obtain their tn after they acquired their last locks and before

[9] A lock point of a transaction is any point in time between the last lock acquired and the first lock released

[10] i.e., a version created by a still active transaction

Operation Invocation	Operation Execution
$begin(T)$	ϕ
...	
$read(x)$	get read-lock on x /*may wait according to the 2PL protocol*/ return the most recent version of x
...	
$write(y)$	get write-lock on y /*may wait according to the 2PL protocol*/ creates a new version of y
...	
$end(T)$	get $tn(T)$ from TM $commit(T)$: perform database updates with version number $tn(T)$ release locks

Figure 5: Execution of a W transaction

Operation Invocation	Operation Execution
$begin(T)$	ϕ
...	
$read(x)$	get read lock on x /*may wait according to the 2PL protocol*/ return the most recent version of x
...	
$write(y)$	get write-lock on y /*may wait according to the 2PL protocol*/ create a new version of y
...	
$lockpoint(T)$	get $tn(T)$ from TM release S locks
...	
$read(z)$	$check\_read(z)$ /*may wait (see EMV2PL protocol)*/ return z's version with largest version number $\leq tn(T)$
...	
$write(t)$	update the last version of t /*this version was created by T before $lockpoint(T)$ */
...	
$end(T)$	$commit(T)$: perform database updates with version number $tn(T)$ release locks

Figure 6: Execution of W|R transaction

committing, tn's are lockpoints. For R transactions, the TM simply guarantees that their sn is smaller than the tn of any active or forthcoming transaction. Thus, an R transaction reads only versions of committed transactions.

3.3 Correctness

In this section, we prove that the EMV2PL protocol guarantees serializability of all transactions.

For the ease of presentation, we denote $sn(r_i[x_k])$ the timestamp used by T_i to select version x_k. If r_i is executed in an R transaction then $sn(r_i[x]) = sn(T_i)$. If r_i is executed in the read part of a W|R transaction then $sn(r_i[x]) = tn(T_i)$. For the purpose of uniformity, we consider as in [AS89] that $sn(r_i[x]) = \infty$ for any other read operations since these reads use locks and always access the latest version of an item.

Theorem : The EMV2PL protocol guarantees serializability of all transactions.

Proof : Since only strict histories are accepted, we consider only committed transactions. Given an history H produced by the EMV2PL protocol, we prove that MVSG(H) is acyclic. Let $ts(T_i)$ be defined as follows:

$$ts(T_i) = \begin{cases} sn(T_i) & \text{if } T_i \text{ is an R transaction} \\ tn(T_i) & \text{otherwise} \end{cases}$$

We show that MVSG(H) is acyclic by showing that for each of its edges $T_i \rightarrow T_j$, $(i \neq j)$, $ts(T_i) < ts(T_j)$. We perform a case analysis on the types of edges.

(i) $T_i \rightarrow T_j$ is an SG(H) edge. That is, $w_i[x_i] <_H c_i <_H r_j[x_i]$. We thus have $tn(T_j) < sn(r_j[x_k])$. If $sn(r_j[x_k])$ is finite, then $tn(T_j) < ts(T_i)$ (see definition of $sn(r_j[x_k])$). If $sn(r_j[x_k]) = \infty$ then T_j acquired a shared-lock on x after T_i released an exclusive-lock. Since $tn(T_i)$ and $tn(T_j)$ are lockpoints, we have $tn(T_i) < tn(T_j)$ (recall that with the S2PL policy, if $T_i \rightarrow T_j$, *all* lock points of T_i precede *all* lock points of T_j). Thus in both cases, $ts(T_i) < ts(T_j)$.

(ii) $T_i \rightarrow T_j$ is a version order edge because for some x and T_k, $r_i[x_k] \in H$ and $x_k \ll_x x_j$. If $sn(r_i[x_k]) = \infty$ then T_i acquired a shared-lock on x. Also T_i acquired its lock before T_j acquired an exclusive-lock on x (otherwise, T_i would have read x_j instead of x_k). We have then $ts(T_i) < ts(T_j)$ since $ts(T_i)$ and $ts(T_j)$ are lock points of T_i, T_j respectively. Suppose now that $sn(r_i[x_k])$ is finite. If $r_i[x_k] <_H w_j[x_j]$, then $ts(T_i)$ was obtained before $ts(T_j)$ and it follows that $ts(T_i) < ts(T_j)$. Finally if $w_j[x_j] <_H r_i[x_k]$, then we must have $tn(T_j) < sn(r_i[x_k])$, otherwise we would have obtained $w_j[x_j] <_H c_j <_H r_i[x_j]$ because the *check_read* function would have blocked T_i until T_j commits.

(iii) $T_i \rightarrow T_j$ is a version order edge because for some x and T_k, $r_k[x_j] \in H$ and $x_i \ll_x x_j$. Since $x_i \ll_x x_j$, T_i acquired an exclusive-lock on x before T_j, so we have $ts(T_i) < ts(T_j)$. □

3.4 Deadlocks

Clearly, EMV2PL suffers from deadlocks since it uses S2PL for serializing W transactions and the write parts of W|R transactions. However, once a W|R transaction starts executing its read part, it may not be involved anymore in deadlocks. This is easily shown as follows.

Suppose T_i is a W|R transaction which executes its read part. It thus has already obtained its *tn*. A deadlock implies a cycle in the transaction wait-for graph[11]. T_i is involved in a deadlock if it belongs to a cycle or if it is waiting for a transaction from a cycle. We first show that T_i may not belong to a cycle. Let $T_i \rightarrow T_{i_1} \rightarrow T_{i_2} \rightarrow ... T_{i_k} \rightarrow T_i$ be a cycle. Since T_i is waiting for T_{i_1}, then T_{i_1} has also obtained its *tn* and $tn(T_i) > tn(T_{i1})$ (only in this case could *check_read* have blocked T_i). Furthermore, T_{i1} may not be a W transaction because once a W transaction has obtained its *tn* (i.e., has passed its lockpoint), it may not be waiting for another transaction T_{i_2}. Thus, T_{i_1} is a W|R transaction which executes its read part. Applying this for every node of the cycle, we obtain that all the nodes are such W|R transactions and $tn(T_i) > tn(T_i)$, which is a contradiction.

Suppose now that T_i does not belong to a cycle. If there was a path from T_i to a transaction of a cycle, say T_j, then T_j would also be a W|R transaction executing its read part. This is impossible since it belongs to a cycle □.

This result is significant. Indeed, in most database applications, deadlocks are avoided by a careful design of application programs. A common way of avoiding deadlocks is to make transactions access the data in the same order. However, in an active database system, rules and transactions are usually programmed by different programmers. The application programmer is not aware of all the rules. As a result, the resulting transaction may suffer from a poor design with respect to deadlocks.

3.5 External Consistency

Although EMV2PL guarantees serializability, it does not preserve *external consistency*. That is, the order in which transactions commit may differ from their serialization order. For example, suppose we have a transaction *entry_order*, as in the example of Section 2, and a transaction *refresh_items* which increases the quantity of some item by some number. As shown on Figure 7, an *entry_order* transaction might be told that there is not enough items in stock for its order although a transaction just increased (and committed) the quantity in stock for that item so that the order could have been satisfied. Such consistency "faults" are likely to occur if the read parts of W|R transactions are long.

Should external consistency be critical, it may help to show the value of transaction timestamps to users, instead of showing the transaction commit time, since timestamps reflect the serialization order. Intuitively, the timestamp of a W|R transaction indicates at which time the decision to commit or abort was taken (even

[11] Nodes of the wait-for graph are transactions and there is an edge $T_i \rightarrow T_j$ iff T_i is blocked by T_j (see [BHG87])

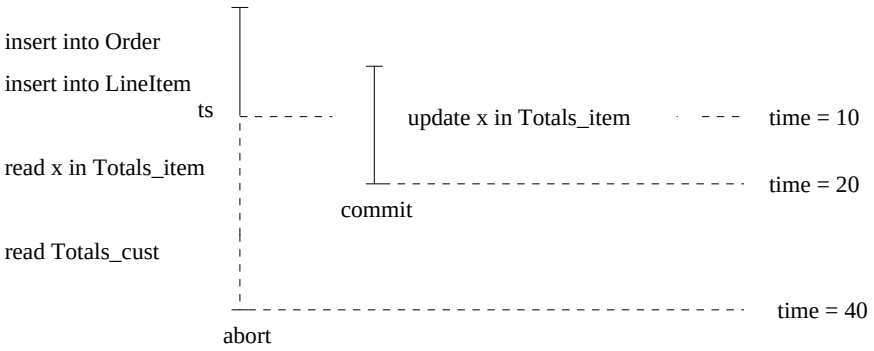


Figure 7: Scenario with an external consistency fault.

though the system committed or aborted the transactions at some later time). For example, the timestamp of the *entry_order* transaction in Figure 7 indicates that there was not enough items at time 10, while the timestamp of *refresh_items* indicates that the increase was completed at time 20.

3.6 Implementation Issues

The EMV2PL protocol may use the same mechanisms as MV2PL to maintain timestamps and versions. Therefore, we mainly discuss here the implementation of the *check_read(x)* function. All the information needed by *check_read*(x) are found in the lock table. Indeed, checking if there is an uncommitted version of x only requires to check if there is an exclusive lock taken on x. If *check_read*(x) must wait for x's creator to commit, it simply waits for the release of this lock. We show that these mechanisms are easily implemented on top of an existing MV2PL lock manager.

We take a lock manager similar to the one described in [GR93]. For simplicity we only consider exclusive locks (noted X-lock) and shared locks (noted S-lock). The lock manager's data structures are summarized in Figure 8. Its two basic interfaces are *lock* and *unlock*. The only change in the lock manager's data structures consists of associating to each lock header a list of process ids, referred to as *list_pid*. This list is used to store the pid(s) of W|R transactions blocked by *check_read*.

Given this, *check_read*(x) performs the following. It looks at the lock header associated with x and checks if x is X-locked. If yes, there is an uncommitted version of x and *check_read*() checks if the owner of this lock has a *tn* smaller than the caller's one. In that case, *check_read*() appends the caller's pid into the lock header *list_pid* and waits for the release of the X-lock. After it woke up, or if it was not blocked, *check_read* returns an ok message. These operations require neither to traverse the lock request queue nor to insert a new lock request into the lock request queue. [MHLP91] proposes a method to check that a page only contains committed data *without inspecting the lock table*. This method can greatly reduce the locking overhead. It can be used in our protocol as follows: *check_read*(x) first checks if the page

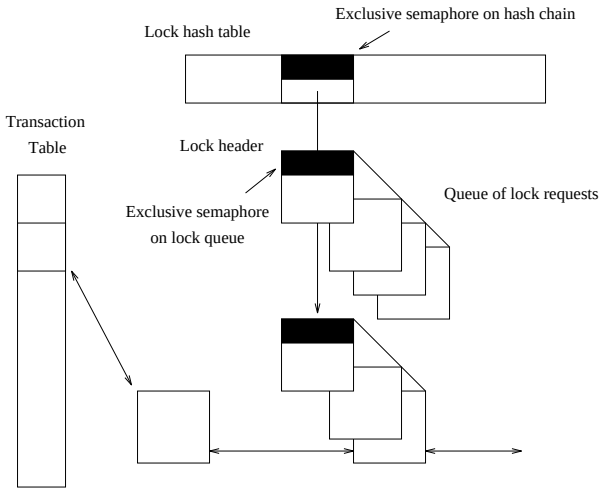


Figure 8: Lock manager data structures

containing x contains committed data as described in [MHLP91]. If this condition is verified, the lock table must not be read.

Finally, we slightly modify the *unlock* interface so that (i) a transaction releasing an X-lock wakes up all processes whose pids are in the lock header *list_pid* and set *list_pid* to null and (ii) a transaction may release all its S-locks. These are the only modifications of the unlock routine.

Note that the *lock* interface is unchanged. In fact, *check_read* has less overhead than *lock*[12] in terms of lock queue traversal and memory space, since instead of enqueuing new lock requests, it (sometimes) appends the caller's pid into a list. The number of locks in the system is thus reduced. We give in Appendix 1, the pseudo-code for *check_read*, assuming the implementation context described in [GR93].

4 Application to Integrity Checking

In this section, we describe how EMV2PL can be used to optimize the enforcement of semantic integrity constraints in database applications. Until now, the vast majority of database applications implement integrity constraints using a *procedural approach* whereby integrity checks are embedded into application programs. In contrast, the *declarative approach*, consists of defining integrity constraints in the database schema, which are then automatically enforced by the database system when needed. We present each approach, compare different methods for programming integrity checks, and analyze the consequences of each method on the pattern of transactions. Hence, we relate the optimizations enabled by our EMV2PL protocol.

[12] which would be invoked instead of *check_read* in MV2PL or S2PL

4.1 Procedural Approach

The most general method is to check integrity after updating the database, which we refer to as the *write-then-check* method. The granularity of the database update after which integrity checks occur may vary from a single to a set of data modification statements, possibly wrapped into a procedure or a transaction. Sometimes, checking at the end of transactions is mandatory because some temporary inconsistent state is unavoidable during the execution of the transaction, or the interactive effects between two or more updates have to be controlled afterwards, or the integrity checks depend on the logic of the transaction program (specially when conditional branching is used). For integrity checks that only involve database reads, checking at the end of transactions naturally yields W|R transactions. Hence, provided that the programmer has the ability to manually insert a lockpoint (e.g., using a specific command in the transaction program) at the end of their write parts, transaction's execution can take advantage of our EMV2PL protocol.

However, according to the experience of the authors and others ([Sha], [AD], [Coc], [Moh]), a frequent method used by application developers to optimize the enforcement of integrity constraints is to check the parameters of the update commands and their potential effect on the database before actually applying the changes to the database[13]. We call this the *check-before-write* method. Here again, the method can be applied to a single or a set of data modification statements. Checking integrity at the beginning of transactions is expected to bring the following advantages: (i) exclusive locks on the updated data items are held for a shorter time if the updates occur at the end of the transaction, and (ii) less work is possibly wasted when the transaction violates data integrity since there are no unnecessary writes. For integrity checks that only involve database reads, checking at the beginning of transactions naturally yields *read-then-write* transactions, noted R|W transactions. Otherwise, the method yields W transactions. Notably, in client-server applications, where it is common to apply the check-before-write method to the design of stored procedures that embed database update operations (e.g., order an item to a supplier), a transaction that has several stored procedure calls results in a general W transaction. Finally, one must note that the check-before-write method is not always applicable for the reasons explained earlier.

4.2 Declarative Approach

Facilities to define integrity constraints by means of assertions and triggers are now provided with some restrictions by most commercial database systems and constitute a prominent feature of the SQL3 standard, currently under development. We present these facilities and indicate the possible optimizations brought by EMV2PL.

[13] Different programming techniques are possible using program variables and temporary relations, which are not discussed here.

4.2.1 Declarative Assertions

Declarative assertions include two forms of constraints: *check constraints* and *referential constraints*. Referential constraints are definable by means of *foreign key clauses*. A referential constraint involves a referencing relation and a referenced relation. One or several columns of the referencing relation are specified as *foreign key*. The value of these columns must also appear in the appropriate columns of the referenced relation. Insertions and updates to the referencing relation that violates the constraint are disallowed, i.e., the modification is rejected. However, the treatment of updates and deletes to the referenced relation can be specified in the foreign key clause using a *foreign key action* that describes the action to take in case of integrity violation. A "cascade" action propagates the update or the delete to the referencing relation, a "set null" (or "default") action sets to the null (default) value all foreign key columns whose values are no longer matched in the referenced relation, and a "no action" disallows the violating update or delete.

Referential constraints can be checked either immediately after an SQL statement (*immediate* mode[14]), or at the end of the transaction (*deferred* mode). For example, following the SQL-92 and SQL3 standards, a referential integrity constraint may be specified as "deferred", in which case it is executed at commit time, or it can be specified as "deferrable", in which case the transaction dynamically chooses to defer it at commit time by setting its *constraint mode* to "deferred" [SQL90].

The EMV2PL protocol optimizes update transactions that check referential constraints in two ways. When the transaction is a W|R transaction:

- the read locks taken in the "W" part of the transaction by the verification of immediate referential constraints either with a "no action", or due to updates or insertions of tuples in a referencing relation, can be released at the beginning of the "R" part.
- the read operations, in the "R" part, entailed by the verification of deferred referential constraints[15] with "no action", do not acquire read locks.

To illustrate, we come back to the example of Section 2. Suppose that when the *entry_order* transaction does an insert into *Order* and *Lineitem*, the values of *custkey*, *itemkey*, and *suppkey*, which are foreign keys, are immediately checked in the referenced relations. Then, using EMV2PL, the read locks on relations *Item*, *Supplier*, and *Customer* entailed by referential constraints will be released before executing the "R" part of the transaction.

Since referential integrity checking requires to access a number of tuples equal to the number of updated tuples, the above optimizations may concern a small number

[14] In fact, SQL-92 and SQL3 standard require constraints to be *effectivelly* checked after the statement. This does not mean that the constraint must be physically checked at that time but rather than the checking must have the same effect as if it occured after the statement. The rationale is to rule out non-deterministic behaviors which can occur with "in-flight" constraint checking [Hor92].

[15] The constraint is either defined as "deferred" or as "deferrable" and the transaction has set its constraint mode to "deferred".

of read operations for short update transactions. However, because referential integrity is heavily used, there may be a significant fraction of such transactions in the application workload. Thus, the lock contention resulting from those read operations issued by many transactions is not negligible and may create concurrency control hot spots (as discussed in [Moh90]) if an S2PL protocol is used.

In general a check constraint can be any SQL condition, including multi-table conditions[16]. Unlike referential integrity, check constraints may amount to read many tuples from different relations in the case of multi-table conditions. Mono-table conditions only require to check value restrictions or intra-row values for each updated or inserted row in a given table.

EMV2PL optimizes update transactions that enforce check constraints in a way similar to referential constraints: read locks acquired for verifying immediate check constraints can be released earlier in W|R transactions, and read operations entailed by the verification of deferred check constraints never take locks. However, unlike referential constraints, this optimization can be quite payful for multi-table assertions.

4.2.2 Triggers

A trigger consists of an *event* that causes the trigger to be activated, a *condition* that is checked when the trigger is activated, and an *action* that is executed when the trigger is activated and its condition is true. The triggering event is an insertion, deletion or update applied to a given relation, say R. We say that the trigger is *defined* on R. The condition is an SQL search condition over the database, and the action is an atomic procedure that may contain SQL statements combined with other procedural constructs.

The activation time of a trigger specifies if the trigger is executed before or after its triggering event, whereas an execution granularity defines how many times the trigger is executed for the event. Triggers that execute before their event are called *before-triggers*, and those that execute after their event are called *after-triggers*. We assume, as SQL3 does, that before-triggers cannot modify the database using update, delete, or insert statements. Finally, we assume that immediate triggers can either rollback a statement or a transaction, whereas deferred triggers can only rollback the transaction.

As with assertions, EMV2PL optimizes the processing of immediate triggers in W|R transactions by releasing read locks earlier. But unlike deferred check constraints, deferred triggers do not necessarilly yield W|R transactions since the action of triggers can perform database updates. More precisely, the problem is the following: given a transaction ready to commit and a set of deferred triggers activated by the

[16] SQL-92 distinguishes *table check constraints* and *assertions*: A table check constraint is attached to one table and is used to express a condition that must be true for every tuple in the table. An assertion is a standalone check constraint in a shema and is normally used to specify a condition that affects more than one table.

transaction, how can the database system statically detect a transaction lockpoint, i.e., the point between the "W" and the "R" parts (if any)?

In fact, the problem is more complicated because check constraints, referential constraints, and triggers can be mixed together. However, considering the general framework requires to have a precise description of an execution model for triggers and assertions, a still open problem which is largely out of the scope of this paper. Thus, we shall restrict ourselves to the detection of a lockpoint when deferred triggers execute.

A preliminar step is to determine, for each trigger, if:

- the action part of the trigger does not acquire any new exclusive lock, in which case the trigger is said to be an *RCA* trigger[17].
- the trigger cannot activate, directly or transitively, another trigger that acquires new exclusive locks, in which case the trigger is said to be *safe*.

Before formally defining RCA and safe triggers, a few definitions will be useful. Suppose that three relations, ins_R, del_R and up_R are associated with every relation R. They respectively contain the tuples that have been inserted, deleted and updated between the beginning of the transaction and the current state. A tuple of R appears in at most one of these relations, even if it is modified several times. More specifically, a tuple first inserted and then updated appears as being inserted with its updated value, a tuple updated several times appears as being updated with its latest updated value, and the deletion of an updated tuple is considered as a deletion of the original tuple. If a tuple is inserted and then deleted (or vice versa), it does not appear in these tables at all. Thus, ins_R, del_R and up_R together represent the *net effect* of the transaction on R. If I_k is a database state and R a relation, we will note $I_k[R]$ the instances of a relation R in state I_k.

RCA trigger: Let r be a trigger defined on a relation R. Let I_k be the database state just before an operation *op* in the action of r is executed, and I_{k+1} the state resulting from the execution of *op*. If for any state I_k and operation *op* of r's action we have

- $I_{k+1}[ins\_R] \subseteq I_k[ins\_R]$
- $I_{k+1}[up\_R] \subseteq I_k[up\_R]$
- $I_{k+1}[del\_R] \subseteq (I_k[del\_R] \cup I_k[up\_R])$

and if *op* does not modify any instance of a relation other than R, then r is said to be an *RCA trigger*.

Thus, an RCA trigger does not insert new tuples, and does not delete or update tuples that were not previously inserted or updated by the transaction.

[17] RCA stands for Rollback, Compensative, Alerter trigger. Indeed, the action part of triggers that do not acquire new exclusive lock typically (i) perform a rollback, (ii) raise an alert or (iii) overwrite database items that have been already inserted, updated or deleted by the triggering transaction.

Example 3: Suppose we have two relations *Purchase* (custkey, item, quantity, supkey) and *Supplier* (supkey, address), and a constraint saying that "the suppliers specified in *Purchase* must exist in *Supplier*". The following trigger enforces this constraint by undoing deletions to *Supplier* that violate the constraint. In this trigger, the relation *deleted* refers to tuples deleted by the triggering transaction.

```
after delete on Supplier
then  insert into Supplier
      select (supkey, address) from deleted
      where exists (select * from Purchase
                    where supkey = deleted.supkey);
```

In this trigger, *deleted* refers to *del_Supplier*. This trigger is an RCA, i.e., it only inserts tuples that were initially deleted by the triggering transaction.

To determine if a trigger is safe, we use a *triggering action graph* (*TAG*), as defined in [Mel93], whose nodes are triggers and edges represent the "activate" relationship between triggers. If no path goes from an RCA trigger r to a non-RCA trigger, then r is safe. Clearly, triggers whose action part only generates a rollback or raises an alert are safe since they do not change the database. Only triggers that "repair" some data modifications may be both RCA and unsafe.

Example 4: Suppose we have two triggers r_1 and r_2 defined on relation $R1(a1, a2)$. r_1 is activated by updates to a_1, and its action updates attribute a_2 for those a_1's updated tuples. r_2 is activated by updates to a_2 and its action deletes some tuples of a relation R_2. r_1 is RCA yet unsafe because it may activate r_2 which is non-RCA.

We now sketch out how the rule manager of an active database system can dynamically detect the lockpoint of a W|R transaction. When the rule manager receives the "end-of-transaction" signal from a transaction, the set of triggers that have been activated is examined. If all these triggers are RCA and safe[18], then the rule manager immediately signals the lockpoint to the Transaction Manager (TM) since only RCA triggers are going to be executed.

Otherwise (i.e., if some triggers are unsafe, or non-RCA), the rule manager considers a first trigger for execution. Usually, the order in which triggers are executed is determined by pre-defined priorities. If this order is total, the rule manager has no other choice than examining, after having executed a trigger, if all the remaining triggers are safe RCA triggers. If so, it signals the lockpoint to the TM. If the order of trigger execution is partial, the rule manager has some freedom to schedule the triggers. Non-RCA and unsafe RCA triggers are scheduled first (as far as possible), until all remaining triggers are safe and RCA. At that point, the rule manager signals the lockpoint of the transaction to the TM.

[18] This is supposed to be determined at the time triggers are defined.

4.3 Procedural versus Declarative Approach

The declarative approach offers several advantages over the procedural approach (see also [WC96] and [SKD95] for more details). First, the maintenance of integrity constraints is facilitated. Since assertions and triggers are modular, adding a constraint amounts to defining new assertions or triggers that will automatically be invoked by application programs when necessary. On the contrary, adding (changing, or removing) a constraint in application programs requires to change existing application code. As reported in [SKD95], a consequence for very large applications is that the number of integrity checking procedures invoked from succeeding releases of application programs uses to increase monotonically. Calls to these procedures are rarely removed from transactions, though it turns out that changes in the data acquisition process have made some controls obsolete. Discovering such situations requires a lot of effort usually not considered as deserving in individual application programs.

Second, assertions and triggers are *reliable* since they are automatically invoked whenever an appropriate data modification is issued by a transaction. This provides a safe way to ensure that every application obeys specific constraints, regardless of the method used to access the database. On the contrary, with the procedural approach, the correct enforcement of constraints is guaranteed only if every single transaction implements it correctly. This makes data quality dependent on the reliability of programmers and programming methodologies and may be the reason of severe inconsistencies as to the enforced policies. As noted by [CPM96], since an increasing number of applications are being developed by small, autonomous groups of developers with narrow views of the overall enterprise, the enterprise information system is very vulnerable to integrity violations.

The main advantage of the procedural approach is that one can precisely control and tune the performance of integrity checks within each individual transaction. In [SKD95], the authors show that this is a major reason why an application is usually less efficient when it is developed with the declarative approach instead of the procedural approach. This is not possible with the declarative approach for two main reasons. First, a trigger or an assertion is defined in a unique way whatever is the transaction that will invoke it. It is therefore not possible to specialize an integrity check for a specific transaction. Furthermore, for triggers, the activation time (immediate before or after, or deferred) is also determined once for all when the trigger is defined. The "deferrable" mode gives slightly more flexibility since it enables to change the activation time of an assertion (either immediate or deferred) in a particular transaction. Second, the optimization offered by the check-before-write method cannot be implemented using the declarative approach. For instance, it is not possible to specify that an assertion or a trigger must be activated just after a "begin transaction" command and checked prior to the execution of updates that occur in the transaction. Immediate before-triggers are the only exception for single update statements.

Both the procedural and the declarative approaches can benefit from the use of EMV2PL, as shown by Table 1 that summarizes the patterns of transactions

	procedural	declarative
R|W	check-before-write for transactions granularity	none
W|R	write-then-check for transactions granularity	deferred checking : deferred or deferrable assertions, and deferred safe RCA triggers
W	- check-before-write or write-then-check for update granularity - others	- immediate checking : immediate assertions and triggers - others

Table 1: patterns of transactions and integrity checking methods

induced by the implementation methods presented before. However, lockpoints can be detected automatically in the declarative approach (as explained before), whereas they have to be explicitly set in transactions in the procedural approach.

5 Performance Study

5.1 Objectives

The experiments presented in this section have been designed with several objectives in mind.

Objective 1: concurrency versus version overhead

A multiversion scheme exposes to a well known tradeoff between the increased concurrency and the I/O and storage overheads caused by the use of versions. This tradeoff was previously studied in [CM86] and [BC92a] in the case of multiversion two phase locking when read-only transactions execute concurrently with update transactions. In the case of EMV2PL, the problem is complicated by the fact that write-then-read transactions use versions and conflict between each other. Thus, our first objective is to analyze how EMV2PL behaves with respect to the above tradeoff for different transaction workloads that combine both W and W|R transactions.

Objective 2: application to integrity checking

There are many situations where both the check-before-write and write-then-check methods are applicable to program transactions. For instance, in Section 2, we assumed that integrity checks were performed at the end of transaction *entry_order*, thereby conforming to the write-then-check method. However, since the two updates on *Order* and *Lineitem* are independent, the check-before-write method could be used instead. As shown by Table 1, the check-before-write method yields R|W or W transactions which cannot be executed under EMV2PL. Hence, our first interest is to study if the use of EMV2PL for executing W|R transactions yielded by the (procedural) write-then-check method or the (declarative) deferred checking (see Table 1) can counterbalance the optimization brought by the check-before-write method under S2PL.

Suppose now that we are interested in some integrity checks that only involve database read operations, then as indicated by Table 1, (declarative) immediate checking yields W transactions while (declarative) deferred checking yields W|R transactions. Hence, our next interest is to compare the performance of immediate checking under S2PL with the performance of deferred checking under EMV2PL. This comparison is attractive because deferred checks are usually considered to be less efficient than immediate checks while immediate checks may complicate the activity of tuning and maintaining database applications [SKD95].

5.2 Multiversion Background

In this section, we briefly review the CCA scheme ([CFL+82]) and the on page version caching ([BC92a]). We decided to simulate the on-page version caching technique because it is one of the most efficient technique for maintaining version and processing transactions. For simplicity, we shall call both read-only transactions and W|R transactions "readers" since with EMV2PL, both kinds of transactions read versions.

In the CCA scheme, versioning is done at the page level and the database is divided into two parts: a main segment and a version pool. The main segment contains the current version of pages, and the version pool contains their prior versions in reverse order. The version pool is a log-like circular buffer. This scheme has several advantages. First, updates are performed in place and current versions are always clustered. Second, writes in the version pool are sequential and thus efficient. Finally, garbage collection of obsolete versions merely results from the sequential nature of the version pool. The two major disadvantages of this approach are that (i) a long running reader may prevent the garbage collection of version pages and (ii) I/Os to the version pool are expensive as one I/O may be required for each version in the list of chained versions of a page. As a consequence, readers may start to trash if they are sufficiently long. To alleviate these problems, a refine scheme is proposed in [BC92a].

First, versions are maintained for records (instead of pages). Second, a small portion of each page is used for caching previous versions of records. As a result readers may find the adequate version without performing any additional I/Os. Also, these versions may sometimes be eliminated while still in the page and thus have not to be appended into the the version pool at all. With on-page caching, when a record is updated (or deleted), an attempt is made to append it into the cache. If the cache is full, garbage collection is attempted on the cached versions. Intuitively, a version X(t) can be discarded if no reader might possibly read it. This is easily checked by looking at the timestamp of the oldest active reader : if this timestamp is greater than the timestamp associated with the version that immediately precedes X(t), then X(t) can be discarded. Consider the example of Figure 9. If the oldest active reader has timestamp 15, then versions $X(5)$ and $X(0)$ of record X can be discarded. If the oldest active reader has timestamp 25, $X(10)$ can also be discarded.

If garbage collection frees a slot, the prior version of the updated record is appended into the cache. If garbage collection was unsuccessful, then one or several

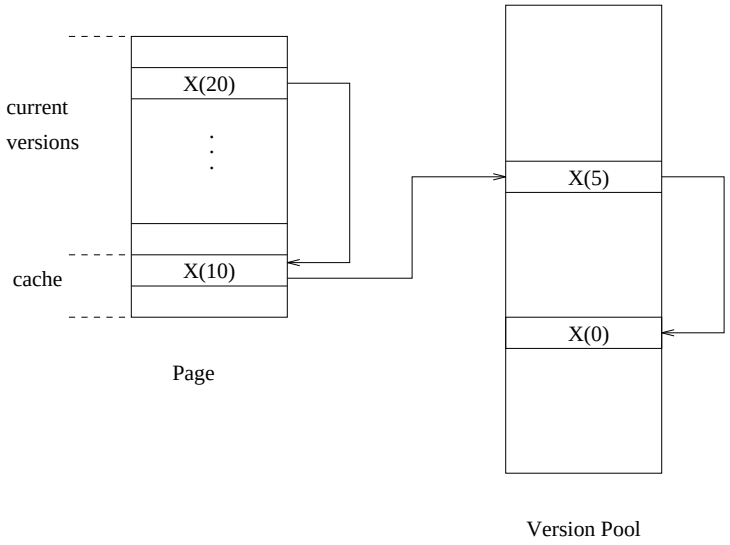


Figure 9: Chained versions of a record X. The timestamp is written between parenthesis.

prior versions in the cache are chosen for replacement and moved to the version pool. With the *write-one* policy, only one version is moved to the version pool. With the *write-all* policy, *all* the cached versions are moved to the version pool at once. In the following we only consider the write-all policy as it is generally more efficient than the write-one policy (see the simulation results of [BC92a]).

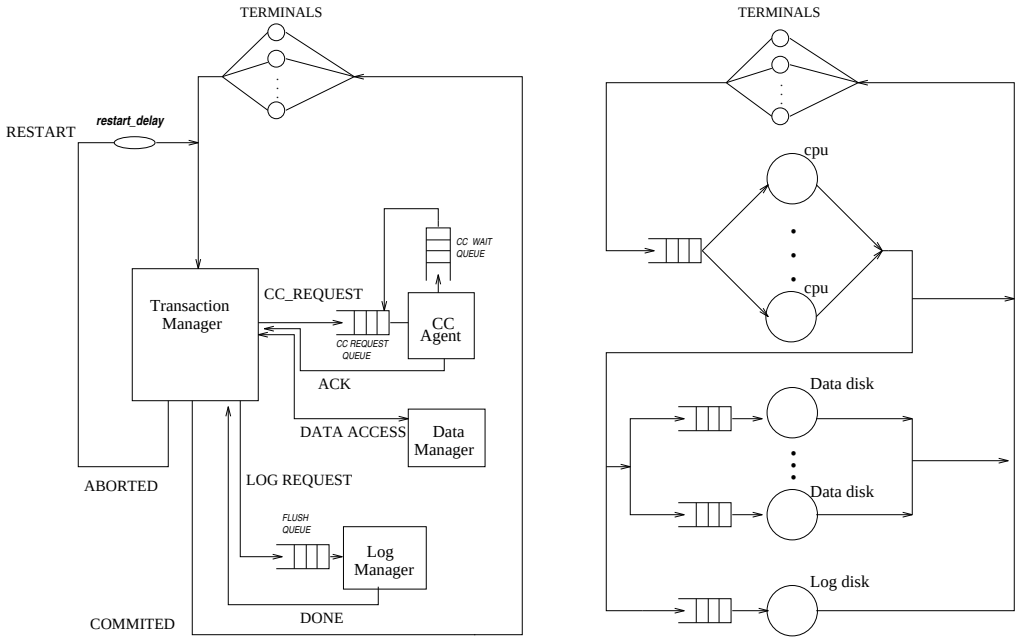
5.3 The Simulation Model

Our simulation model is strongly derived from [CM86], [ACL87], [BC92a] and [SLSV95]. It has two parts: the *system model* simulates the behavior of the various operating system and DBMS components, while the *application model* simulates the database items and the transactional workload.

5.3.1 The System Model

In our simulation, we model the concurrent execution of transactions on a single site database. To keep the simulator simple, we simulate page-level locking. This allows us not to simulate indexes and index locking, and transactions access records randomly.

The system model (Figure 10a) is divided into four main components: a Transaction Manager (TM), a Concurrency Control Manager (CCM), a Data Manager (DM) and a Log Manager (LM). The TM is responsible for issuing concurrency control requests and their corresponding database operations. It also assures the durability property by flushing all log records of committed transactions to durable memory. The CCM schedules the concurrency control requests according to either the S2PL



(a) Logical Queuing Model (b) Physical Queuing Model

Figure 10: The Simulation Model

or EMV2PL protocol. The LM provides read and insert-flush interfaces to the log table. The DM is responsible for granting access to the physical data objects and executing the database operations.

The DM encapsulates the details of a LRU Buffer Manager. The number of pages in the buffer cache is *num_buffer*. These pages are shared by the main segment and the version pool. When a dirty version pool is chosen for replacement by the LRU algorithm, the DM first checks if it contains needed versions. If not (i.e., if it contains only obsolete versions), the page is considered non-dirty and simply discarded. Otherwise, it is written on disk.

The physical queuing model is shown on Figure 10b. There are k resource units, each containing one CPU server and two I/O servers ([CM86]). The requests to the CPU queue and I/O queues are serviced FCFS (first come, first serve). Parameter *record_cpu* is the amount of CPU time for accessing a record in a page. Parameter *page_io_access* is the amount of I/O time associated with accessing a data page from the disk. We added one separate I/O server dedicated to the log file. The parameter *log_disk_io* represents the fixed I/O time overhead associated with issuing the I/O. Parameter *log_rec_io_w* is the amount of I/O time associated with writing a log record on the Log disk in sequential order. Parameter *commit_cpu* is the amount of CPU time associated with executing the commit (releasing locks, etc). Parameter *abort_cpu* is the amount of CPU time associated with executing the abort statement

Parameter	Description	Value
num_buffer	Number of pages in the buffer pool	150
k	Resource Unit (kCPUs and 2k Disks)	1, 2 or 3
page_cpu	CPU time for accessing a record	10 millisecond
page_io_access	I/O time for accessing a page	35 milliseconds
log_disk_io	time for issuing a I/O log access	35 milliseconds
log_rec_io_w	I/O time for sequentially writing 1 page on log disk	1 milliseconds
commit_cpu	cpu time for executing a commit	10 milliseconds
abort_cpu	cpu time for executing an abort	10 milliseconds
restart_delay	restart delay of an aborted transaction	5 milliseconds
cpu_cc_request	cpu time for servicing one cc_request	1 millisecond

Table 2: System Parameters Definitions and Values

(executing undo operations, releasing locks etc). Table 1 summarizes the parameters of the system model and their values for the experiments.

5.3.2 The Application Model

The database contains *num_rec* Wisconsin benchmark-sized records ([Gra91]). With S2PL, 36 records fit in one page (this corresponds to pages of 8K containing records of 227 bytes). With MV2PL or EMV2PL the records are assumed to contain an additional 8 bytes to store the timestamp and version pointer. As a result, only 34 records fit in one page, whose *num_cached* records are used to cache previous versions.

Transactions are either R|W, W|R or W transactions. We vary the structure of these transactions, i.e., the number of read and write operations and the probability of executing a rollback at the end of the read part of R|W and W|R transactions. When the read part consists of integrity checks, this enables to simulate the detection of an integrity violation that leads to reject the transaction.

A transaction workload consists of a mix of transactions of different kinds. We use the following parameters. There are *num_terms* terminals executing transactions. Parameter *WR_frac* is the fraction of terminals executing W|R (or R|W) transactions. Parameter *W_size* is the average number of operations executed by the W transactions and by the write part of W|R (R|W) transactions. Among these operations, *percent_write_W* (*percent_write_WR*) are write operations. Parameter *R_size* represents the number of read operations executed by the W|R transactions in their read parts. These parameters are summarized in Table 2 (where "transaction" is abbreviated "tx").

Regarding the measurements, each simulation consisted of 3 to 5 repetitions, each consisting of 2000 seconds of simulation time. These numbers were chosen in order to achieve more than 90 percent confidence intervals for our results.

Parameter	Description	Value
num_rec	Number of records in the database	20000
num_cached	Number of records cached in each page	3
num_terms	Number of terminals	20
WR_frac	fraction of W|R tx	-
W_part_size	mean size of W|R tx write part	-
R_part_size	mean size of W|R tx read part	-
percent_write_WR	fraction of write in W|R tx write part	-
W_size	mean size of W tx	-
percent_write_W	fraction of write in W tx	-

Table 3: Workload Parameters Definitions

Parameter	Value
WR_frac	100%
percent_write_WR	50%
W_part_size = *R_part_size*	3 to 7

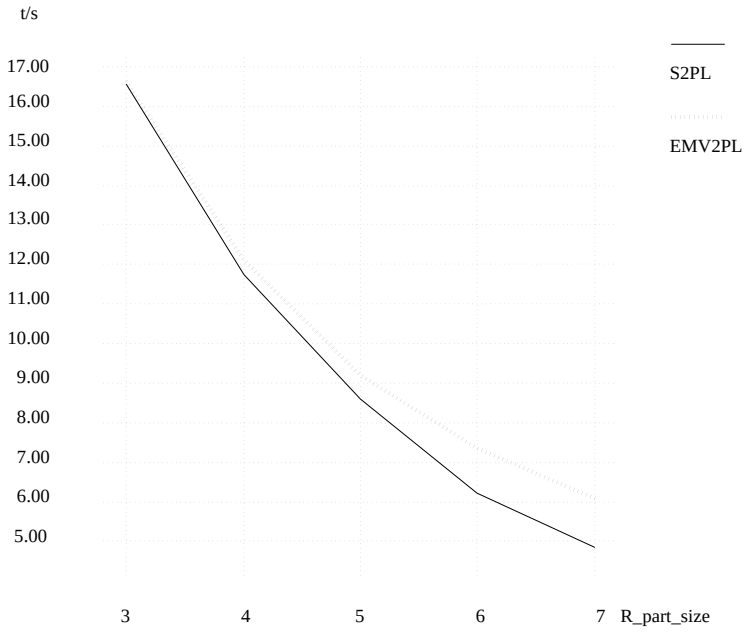


Figure 11: W|R transaction throughput

5.4 Experiment 1

The goal of this experiment is to show the value of EMV2PL for applications containing W|R transactions. The workload contains only W|R transactions and the variable is the size of the transactions. The write part and the read part of W|R transactions contain the same number of operations. In the write part, 50% of operations are write operations. W|R transactions executed with S2PL (resp. EMV2PL) are noted W|R_{s2pl} transactions (resp. W|R_{emv2pl} transactions).

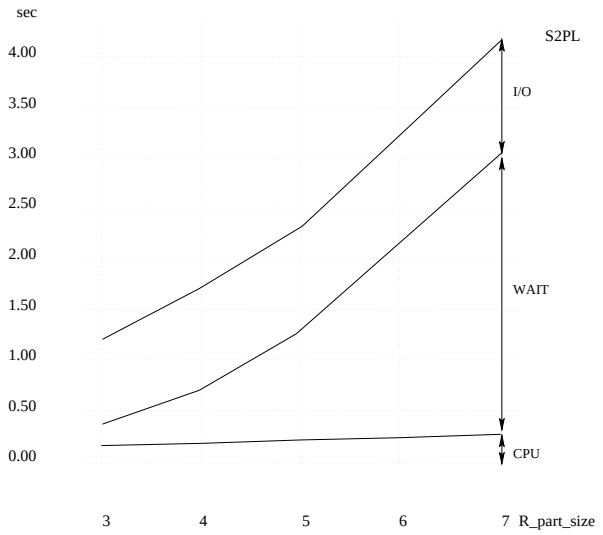


Figure 12 W|R tx response time (S2PL)

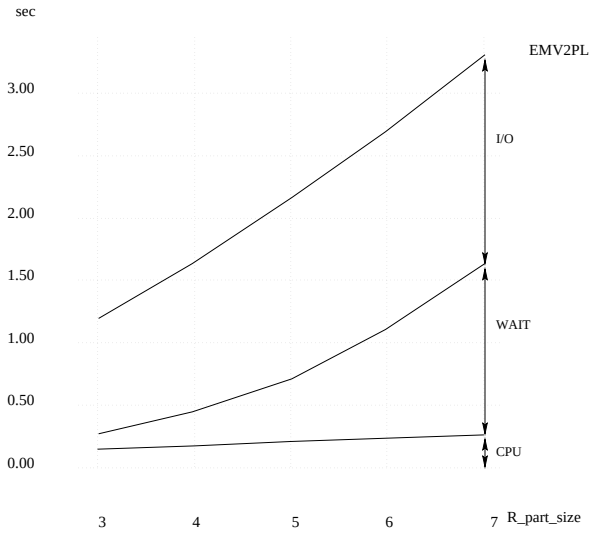


Figure 13: W|R tx response time (EMV2PL)

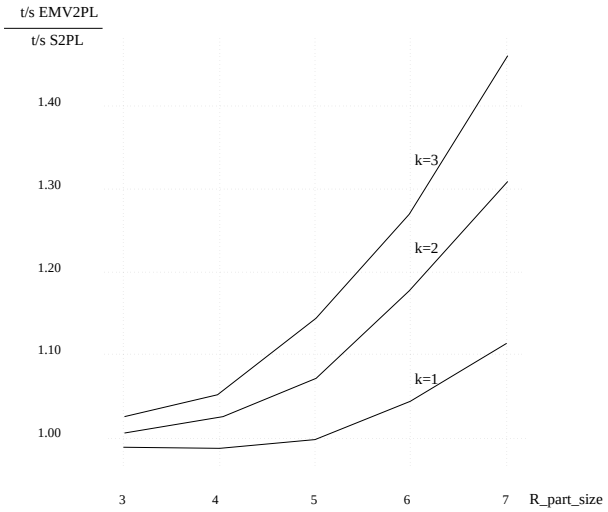


Figure 14: W|R transaction throughput

Figure 11 shows the throughput of W|R transactions. The throughput of W|R$_{emv2pl}$ transactions is always better than the throughput of W|R$_{s2pl}$ transactions. The longer are the W|R transactions, the bigger is the gain in performance. Figure 12 (resp. 13) shows the response time of W|R transactions and how it is splitted into CPU, wait and I/O times under S2PL (resp. EMV2PL). EMV2PL significantly reduces the wait time of W|R transactions while the contention on disk servers increases because transactions execute operations at a faster rate. The number k of resource units is thus an important parameter.

Figure 14 shows the gain in throughput of W|R transactions relative to S2PL with one, two or three resource units. It shows that EMV2PL is more efficient as there are more resource units since the gain in concurrency is less affected by a higher

Parameter	Value
W_size	1
percent_write_W = percent_write_WR	100%
W_part_size	1
R_part_size	5 to 20

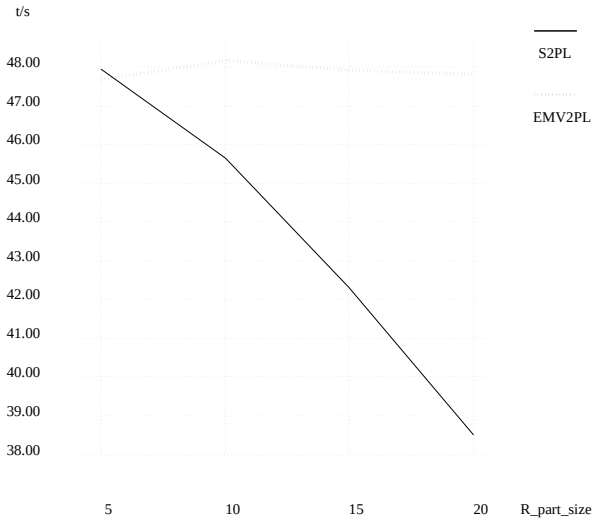


Figure 15: W transaction throughput

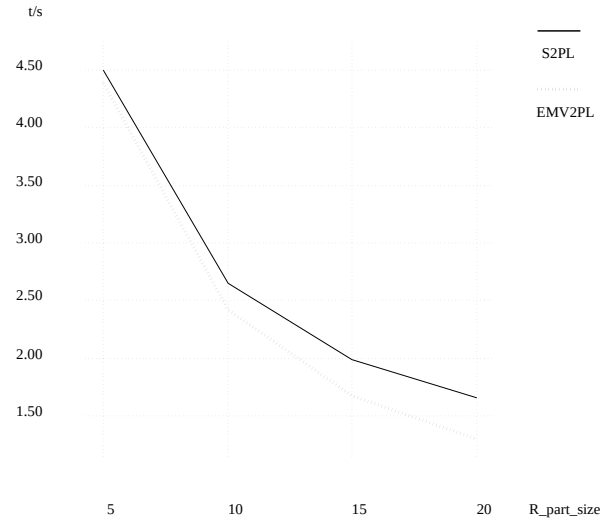


Figure 16: W|R-R|W transaction throughput

contention on disk servers. In this simulation, the maximum abort rate was 12% under S2PL and 0.7 % with EMV2PL, with transactions of size 14. EMV2PL eliminates most deadlocks. These results are achieved with a very low storage overhead[19].

5.5 Experiment 2

In this experiment, we refine the analysis of the previous experiment. The workload consists of W and W|R transactions. W transactions update one record, as does the write part of W|R transactions. We vary the length of the read part of W|R from 5 to 20 operations. *WR_frac* is set to 20 %. The buffer size is held fixed at 150 pages and there are 2 resource units ($k = 2$).

Figure 15 (resp. 16) shows the throughput of W (resp. W|R) transactions. These figures illustrate the well-known tradeoff of multiversion concurrency control. S2PL provides good performance for long W|R transactions: the system resources are largely devoted to the execution of W|R transactions while W transactions are blocked. The longer are W|R transactions, the more W transactions have to wait. This corresponds to the observation made in [BC92b] where long read-only transactions execute concurrently with short W transactions.

In contrast, by completely eliminating read-write blockings between W and W|R transactions, EMV2PL achieves a high throughput for W transactions. Consequently

[19] in fact, W|R transaction almost always read the most recent versions. No version was written back to the Version Pool; they were all eliminated while in the page caches using garbage collection

Parameter	Value
W_part_size = *R_part_size*	6
percent_write_WR	50%
W_size	12
percent_write_W	25%

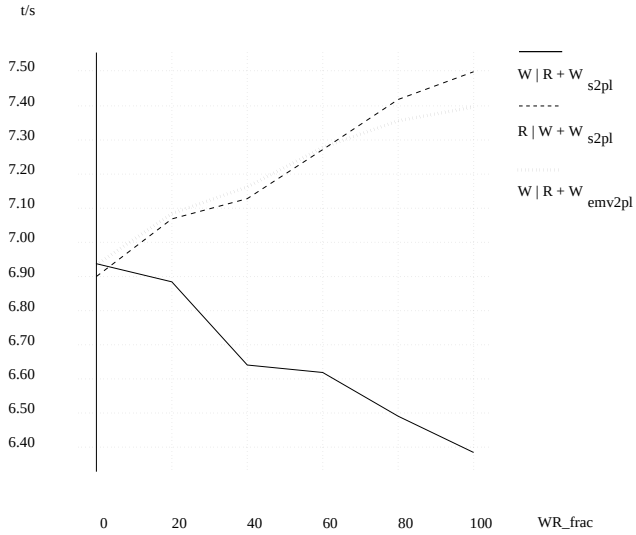


Figure 17: transaction throughput (0% rollback)

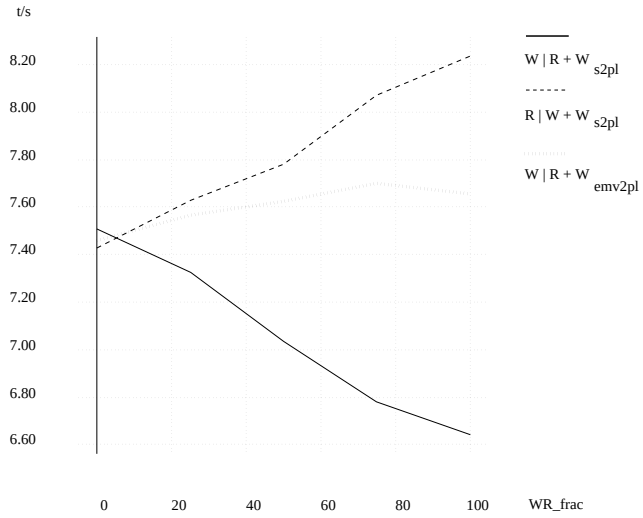


Figure 18: transaction throughput(11.5% rollback)

the throughput of W transactions is unaffected by longer W|R$_{emv2pl}$ transactions. The throughput of W|R$_{emv2pl}$ transactions is below W|R$_{s2pl}$ because the dispute for system resources with W transactions is more intense and in addition, they perform slightly more I/Os for reading versions. These factors counterbalance the gain in concurrency achieved by EMV2PL: for *R_part_size* = 15, a W|R$_{s2pl}$ transaction spends 1.8 seconds in I/Os and waits 0.14 seconds for locks, whereas a W|R$_{emv2pl}$ spends 0.5 more seconds in I/Os and 0.1 less seconds waiting for locks.

5.6 Experiment 3

In this experiment, we evaluate the effectiveness of EMV2PL in workloads containing a varying proportion, WR_frac, of W and W|R (or R|W) transactions. We selected parameters *W_part_size*, *R_part_size*, *percent_write_WR*, *W_size*, and *percent_write_W* so that each W, W|R, and R|W transaction executes 3 writes and 9 reads, among which 6 reads are assumed to model integrity checks. The 6 reads are either placed at the beginning or at the end of the transaction, yielding respectively R|W or W|R transactions, or interleaved with write operations, yielding W transactions. Using these assumptions, W|R transactions model a deferred checking or a write-then-check policy, R|W transactions model a check-before-write policy, and W transactions model all other integrity checking policies (including immediate checking, as shown by Table 1).

Parameter	Value
W_part_size	8
R_part_size	4
percent_write_WR	50%
W_size	12
percent_write_W	33%

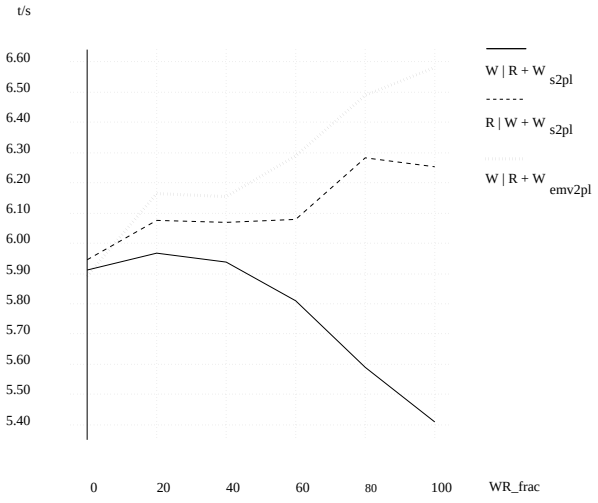


Figure 19: transaction throughput (0% rollback)

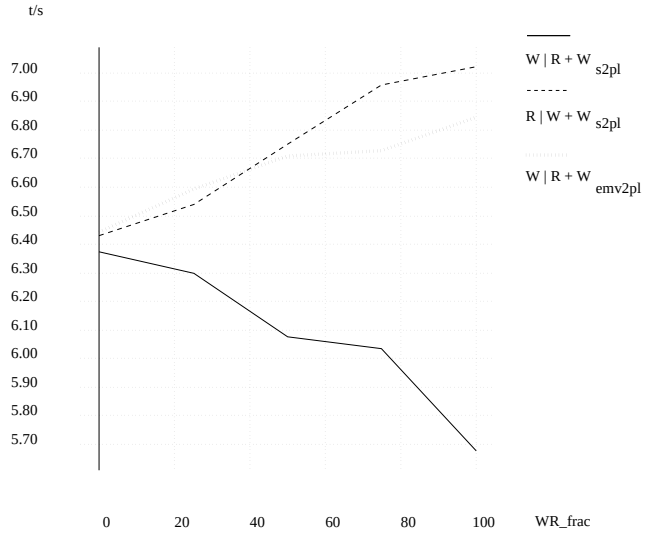


Figure 20: transaction throughput(11.5% rollback)

A workload of W and W|R transactions is executed under S2PL (curve W|R_{s2pl}) and EMV2PL (curve W|R_{emv2pl}), while a workload of W and R|W transactions is executed under S2PL only (curve R|W_{s2pl}). We vary *WR_frac* from 0% to 100%.

Figure 17 shows the total transaction throughput[20] when no integrity check can issue a transaction rollback. Curve R|W_{s2pl} shows that when *WR_frac* increases i.e., more and more transactions execute their checks prior to any update, the throughput increases because there are fewer lock conflicts. Thus, checking integrity at the beginning of transactions provides the best performance result under S2PL. On the contrary, curve W|R_{s2pl} shows that when more transactions check integrity at the end, the throughput decreases due to more lock conflicts. Thus, checking integrity at the end of transactions gives the worst performance result under S2PL.

Interestingly, curve W|R_{emv2pl} shows that with EMV2PL, (i) when more transactions check integrity at the end, the transaction throughput increases, and (ii) the throughput is almost always as good as the throughput for R|W_{s2pl} transactions.

We now show the influence of transaction rollbacks, caused by integrity checks. In a W transaction, a rollback is assumed to occur after every 4 operations in order to simulate the dissemination of checks within the transaction. In a W|R

[20] i.e. the sum of the W transaction throughput and the W|R (or R|W) transaction throughput. Since all transactions have the same number of operations and the same proportion of write operations, the total throughput enables us to directly compare the immediate, deferred and check-before-write strategies.

(or R|W) transaction, a rollback can only occur in the read part after every two read operations. If each check has a probability p to roll back the transaction, the transaction has a total probability of $1-(1-p)^3$ to roll back and $(1-p)^3$ to commit.

Figure 18 shows the transaction throughput when $p = 0.04$, that is, 11.5 %[21] of the transactions roll back. R|W_{s2pl} transactions are now more efficient than W|R_{emv2pl} transactions because they do not execute unecessary operations.

Figures 19 and 20 show the results of the same simulations when each transaction executes 8 reads, among which 4 represent integrity checks, and 4 writes. Thus, *percent_write_W* = 33%. The resulting curves have the same shape as before but the throughput of W|R_{emv2pl} is now higher than the throughput of R|W_{s2pl}, i.e., checking integrity at the end of transactions under EMV2PL is better than checking integrity at the beginning of transactions under S2PL. With a larger fraction of write operations (33% instead of 25%), there are more lock conflicts. This significantly increases the proportion of aborts due to deadlocks for R|W and *W* transactions under S2PL (8% for R|W_{s2pl} at *WR_frac* = 100%) while EMV2PL eliminates most of the deadlocks[22] (2% for W|R_{emv2pl} at *WR_frac* = 100%). Moreover, since there are more lock conflicts, the system resources are less utilized. This situation favours the performance of W|R_{emv2pl} transactions since EMV2PL consumes more resources[23] than S2PL (At *WR_frac* = 100%, W|R_{emv2pl} transactions spend 0.4 less seconds in waiting time than W|R_{s2pl} and spend only 0.10 more seconds in I/Os).

5.7 Experiment 4

In this experiment, we refine the analysis of the previous experiment. We fix *WR_frac* to 100%, so that each workload consists either in W|R or R|W transactions. We keep *percent_write_WR* equal to 50%, and we vary *W_part_size* and *R_part_size* from 3 to 7. Thus, W|R transactions access n records in their write part (from 3 to 7) whose half (in average) are updated, and read n records in their read part. R|W transactions do the reverse.

Figure 21 shows the throughput of W|R_{emv2pl}, W|R_{s2pl}, and R|W_{s2pl} transactions when no transaction rolls back. The time spent by transactions to wait for some locks and to perform I/Os is respectively shown on Figure 23 and 24. Clearly, W|R_{s2pl} are slowed down by lock conflicts (see Figure 23) because their write locks are held for a longer time, while the read part is executed. Hence, lock conflicts between write operations and concurrent read operations are more likely to occur. In comparison, R|W_{s2pl} transactions hold their write locks only during their write part: their waiting time is about 22% smaller than the waiting time of W|R_{s2pl} transactions.

Figure 23 shows that W|R_{emv2pl} transactions suffer less from lock conflicts than R|W_{s2pl}. The difference increases as the transaction is longer because the effect of releasing read locks earlier and accessing versions becomes more effective. However, as shown by figure 24, this gain is completely traded off by a higher contention

[21] $1-(1-0.04)^3$
[22] see section 3.4
[23] because of versions

Parameter	Value
W_part_size = *R_part_size*	3 to 7
percent_write_WR	50%
WR_frac	100%

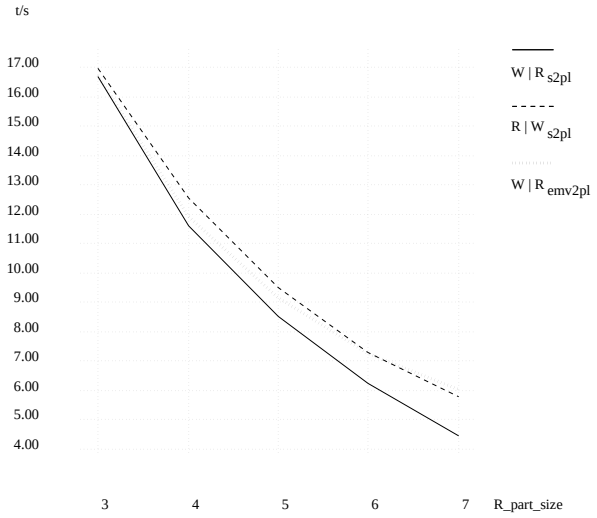


Figure 21: Transaction throughput (0% rollback)

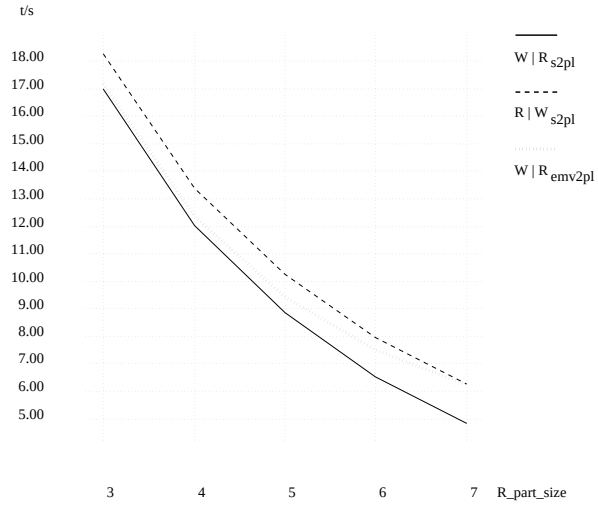


Figure 22:Transaction throughput (10% rollback)

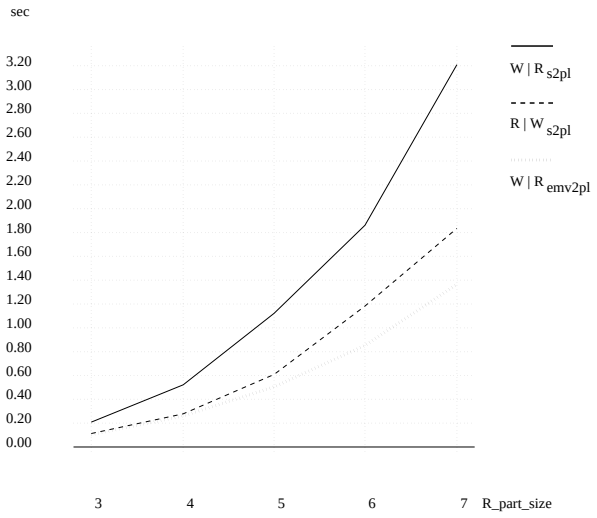


Figure 23: Transaction wait time

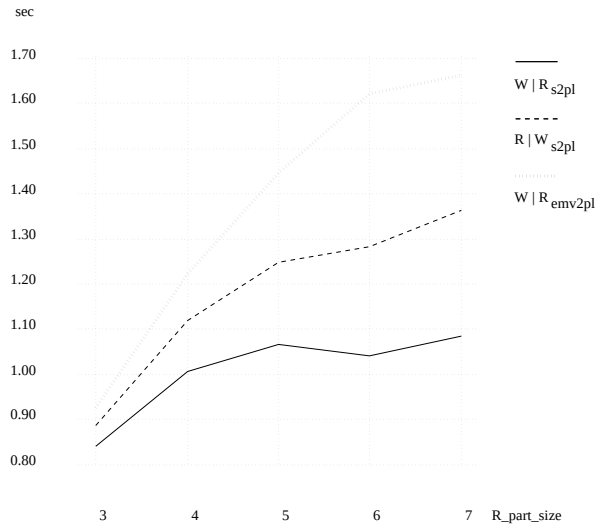


Figure 24: Transaction I/O time

on I/O resources (including the additional I/O cost incurred by the versions). As a result, R|W$_{s2pl}$ transactions exhibit slightly better performance than W|R$_{emv2pl}$ (see Figure 21). Only long W|R$_{emv2pl}$ transactions (*R_part_size* greater than 6) outperform R|W$_{s2pl}$ transactions because R|W$_{s2pl}$ transactions start beeing affected by deadlocks (12%).

We ran the same experiment with more resource units (k = 4): With a lower resource

contention, W|R_{emv2pl} transactions exhibit almost similar performance as R|W_{s2pl} transactions (curves are not shown).

Last, Figure 22 shows the throughput of W|R and R|W transactions when 10% of the transactions roll back. Clearly, R|W_{s2pl} transactions exhibit better performance than W|R_{emv2pl} transactions, which sometimes may execute unecessary update operations.

5.8 Lessons learned

• Concurrency Versus Version Overhead

EMV2PL has some similarities with MV2PL. In a workload where W|R transactions execute concurrently with short W transactions, EMV2PL dramatically reduces the number of lock conflicts between W|R and W transactions, thereby enabling a high throughput for W transactions (to the detriment of W|R transactions). Furthermore, with respect to S2PL, EMV2PL reduces the conflicts and deadlocks between concurrent W|R transactions. If the system resources are sufficiently large, this gain in concurrency yields a significant increase of W|R transaction throughput, and it is achieved with a small utilization of versions.

• Application to Integrity Checking

Consider now integrity constraints, which can possibly trigger a transaction rollback if a constraint is violated, and whose enforcement only requires database read operations. The main result of our experiments is to show that: if the probability that a transaction violates integrity is small (below 10% in our experiments)[24], then checking integrity at the end of transactions run under EMV2PL, often achieves a better total transaction throughput than all other methods. In the other case, checking integrity at the beginning of transactions under S2PL is usually better, which justifies a posteriori the check-before-write method often used by application developers.

As our experiments show, EMV2PL optimizes transactions that check integrity at the end by enabling those checks to use versions, but also by releasing the read locks taken by the transaction earlier. For instance, in the experiment 3, EMV2PL gives a high throughput when the write part of the transactions perform as many read operations as the read part. This is worth noticing because the read locks may be required by integrity checks that involve both read and write operations (e.g, "cascade" immediate referential constraints). As a consequence of the above result, (declarative) deferred checking generally offers better performance that immediate checking for "read-only" integrity checks. However, our modeling only considers the low level read and write operations incurred by integrity checks. We ignore the overhead associated with deferred checking to record the tuples affected by the updates in the write part of W|R transactions[25]. Likewise, we ignore the possibility of grouping database accesses, common to multiple integrity checks, enabled by the deferred checking method. Thus, although EMV2PL relaunches the interest of deferred checking, deeper performance analysis is needed to explore the tradeoff between deferred and immediate checking.

[24]Note that for transaction processing applications, (e.g., in the banking domain), the percentage of integrity errors is reported to be below 1%.

[25]This overhead is one of the declared reasons why database products do not implement deferred checking today.

6 Related Work

An extensive literature addresses the problem of designing concurrency control algorithms that augment the performance of concurrent transactions. Our work directly builds on previous work on MV2PL protocols ([CFL+82], [BHG87], [AS89], [AK91], [BC92a], [MPL92]), but differs from those by focusing on the specific class of write-then-read transactions. In [BC92a] and [MPL92], the authors propose techniques in which an update transaction does not systematically create a new version of its updated items. Hence, read-only transactions do not access versions which are the most up-to-date before their starting time. Instead, they all read a given older state which is periodically refreshed. The goal is to reduce the number of versions stored in the database (see [BD92] and [MWC92] who studied the impact of these techniques). However, this technique cannot be used in our protocol because the read part of W|R transactions must access the most up-to-date versions that precede their starting time. Otherwise, the read part of a W|R transaction may read versions out-of date with respect to the one read by the write part, and a serialization fault may occur.

So far, the largest body of work on the enforcement of semantic integrity constraints has focused on efficient algorithms to detect if a constraint is violated (e.g, [HI85], [BD95], [CW90]). The problem of optimizing the execution of multiple integrity checks within a transaction has been first addressed in [BP79]. This paper compares the performance of different constraint checking policies (including the check-before-write and the write-then-check methods) and show that the check-before-write method is the most efficient (in terms of transaction response time) because it avoids redundant computations and expensive rollbacks. However, all these works do not consider the possible concurrency between transactions.

A very few research papers have addressed the problem of optimizing the throughput of concurrent transactions that perform integrity checks. The *Commit_LSN* method, proposed in [Moh90], is used (among other things) to avoid taking a lock when checking a referential integrity constraint. More precisely, no read lock is acquired on data items involved in a "no action" referential integrity constraint [26] if (i) the constraint is satisfied, and (ii) a property of the *Commit_LSN* is verified. In contrast, our proposal is not limited to referential constraints (verified or not) but can be applied to any integrity check as long as it does not require new write locks. However our method only applies to W|R transactions.

In [Laf82], the author proposes to use semantic integrity dependencies between data items to improve the efficiency of constraint checking. The checking of data, which "depend" on the data items currently updated by a transaction, is delayed until other operations actually need them. This gives the possibility to perform these checks in parallel, thereby improving the concurrency of transactions. In the field of active databases, a related idea has been proposed in [DHL90], which consists of executing triggers in separate transactions from the triggering transaction. When a trigger is defined, a *decoupled* coupling mode can be chosen, indicating that the

[26] immediate or deferred

trigger is run in a separate transaction. This mode is subdivided into *dependent decoupled* where the separate transaction is not spawned unless the triggering transaction commits, and *independent decoupled*, where the separate transaction is spawned regardless of weather the triggering transaction commits. As shown in [CJL91], defining triggers in a decoupled mode may greatly improve the throughput of concurrent transactions. However, these two methods require some semantic analysis in order to guarantee that no transaction sees a temporary inconsistent database state. In constrast, our solution preserves the full isolation of all transactions.

7 Conclusions

EMV2PL is a simple yet efficient extension of MV2PL which enables a W|R transaction that has acquired all its write locks to (i) release its read locks, and (ii) execute new read operations on versions without taking locks. We proved the correctness of this protocol, and showed that its implementation only requires a few changes with respect to an existing implementation of MV2PL. Performance studies show that for workloads containing W|R transactions, EMV2PL can significantly improve the overall throughput of transactions (i.e., W|R, and W transactions), with a relatively small utilization of versions.

We then presented a specific, yet important, application of our protocol to the problem of integrity checking. We described various possible methods for implementing integrity checking. For "read-only" integrity checks, we showed that: if the probability that a transaction violates integrity is small, then checking integrity at the end of transactions run under EMV2PL, is the method that often achieves the best total transaction throughput. Hence, (declarative) deferred checking generally offers better performance that immediate checking for "read-only" integrity checks, which in our view relaunches the interest of implementing deferred assertions and deferred triggers in relational database systems.

We foresee two directions of future work. One is to extend our simulation experiments to handle non uniform lock conflicts between data items. An interesting application of this is given by "summary tables", which consist of materialized views computed from base relations. Relations *Totals_item* and *Totals_cust* in the example of Section 2 are two examples of summary tables. These tables are quite frequent in decision support applications and for integrity checking. We plan to investigate the performance of EMV2PL in application scenarios where summary tables are read at the end of transactions. Another direction is to compare more in depth the performance of immediate versus deferred checking by taking into account the respective overheads associated with these two methods.

Acknowledgments

We are grateful to Anthony Tomasic for his detailed comments that enabled to improve this paper. We also thank Françoise Fabret, Angelika Kotz-Dittrich, C. Mohan, and Dennis Shasha for constructive discussions about the paper.

References

[ACL87] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Computers and Systems*, 12(4):609–654, December 1987.

[AD] A.Kotz-Dittrich. private communication.

[AK91] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, Colorado*, 20:98–107, May 1991.

[AS89] D. Agrawal and S. Sengupta. Modular synchronisation in multiversion databases: Version control and concurrency control. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Portland, Oregon*, pages 408–417, 1989.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, San Jose, California*, pages 1–8, May 1995.

[BC92a] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. *Proc. Int. Conf. on Data Engineering, Tempe, Arizona*, pages 535–545, February 1992.

[BC92b] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. *Proc. Int. Conf. on Data Engineering, Tempe, Arizona*, pages 535–545, February 1992.

[BD92] P. Bober and D.M. Dias. Storage cost tradeoffs for multiversion concurrency control. Technical Report RC 18367, IBM Research Division, T.J. Watson Research Center, July 1992.

[BD95] V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *The International Journal on Very Large Databases*, 4(3):493–518, July 1995.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BP79] D.Z. Badal and G.J. Popek. Cost and performance analysis of semantic integrity validation methods. *Proc. ACM SIGMOD Int. Conf. on management of Data, Boston, Mass.*, pages 109–115, 1979.

[CFL+82] A. Chan, S. Fox, W.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. *Proc. ACM*

SIGMOD Int. Conf. on Management of Data, Orlando, Florida, pages 184–191, June 1982.

[CJL91] M. C. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases : A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3), September 1991.

[CM86] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computers and Systems*, 4(4):338–378, November 1986.

[Coc] B. Cochrane. private communication.

[CPM96] R.J. Cochrane, H. Pirahesh, and N. Mattos. Integrating triggers and declarative constraints in sql database systems. Technical Report 4861/RJ9989, IBM Almaden, 1996.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. *Proceedings of the 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia*, pages 566–577, 1990.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, New Jersay*, pages 204–214, May 1990.

[GR93] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufmann, 1993.

[Gra91] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing systems.* Morgan Kauffmann, 1991.

[HE91] L. Hobbs and K. England. Rdb/vms, a comprehensive guide. *Digital press*, 1991.

[HI85] A. Hsu and T. Imielinski. Integrity checking for multiple updates. *Proc. of the Int. Conf. on Management of Data, Austin, Texas*, May 1985.

[Hor92] B.M. Horowitz. A run-time execution model for referential integrity maintenance. *Proceedings of the Eight Int. Conf. on Data Engineering, Tempe, Arizona*, February 1992.

[Ill94] Illustra Information technologies, Oakland, CA. *Illustra user's guide*, 1994.

[Laf82] G.M. Lafue. Semantic integrity dependencies and delayed integrity checking. *Proc. of the 8th Int. Conf. on Very Large Database Systems, Mexico City, Mexico*, pages 292–299, 1982.

[Mel93] J. Melton, editor. *(ISO/ANSI Working Draft) Database Language SQL3.* Number ANSI X3H2-90-412 and ISO DBL-YOK 003. February 1993.

[MHLP91] C. Mohan, D. Haderle, B.G. Lindsay, and H. Pirahesh. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transaction on Database Systems*, 17(1):94–162, 1991.

[Moh] C. Mohan. private communication.

[Moh90] C. Mohan. Commit_lsn: a novel and simple method for reducing locking and latching in transaction processing systems. *Proc. of the 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia*, pages 406–418, August 1990.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *Proc. ACM SIGMOD Int. Conf. on Management of Data, San Diego, California*, pages 124–133, June 1992.

[MWC92] A. Merchant, K.-L. Wu, and M.-S. Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query-processing. *Proc. of ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, Newport, Rhode island*, 20(1):103–114, June 1992.

[Par89] Part 800-V1.0, Oracle Corp. *PL/SQL User's Guide and Reference, Version 1.0*, 1989.

[Sha] D. Shasha. private communication.

[SKD95] E. Simon and A. Kotz-Dittrich. Promises and realities of active database systems. *Proc. of the 21st Int. Conf. on Very Large data Bases, Zurich, Switzerland*, pages 642–652, September 1995.

[SLSV95] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3), December 1995.

[SQL90] Iso /ansi sql2. working draft, October 1990.

[Tha94] M. Thakur. Transaction models in interbase 4. *Proc. of the Borland Int. Conf.*, June 1994.

[TPC95] Tpc benchmark^{TM} D. (Decision Support), Standard Specification, Revision 1.0, May 1995.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Fransisco, 1996.

Appendix 1

The code of the *check_read* function follows (see [GR93] for a precise description of the data structures used here). For simplicity, only the shared (S) and exclusive (X) modes are considered.

```
lock_reply check_read(lock_name name)
{
long bucket;                                        /*index of hash bucket*/
lock_head* head;                                    /*pointer to lock header block*/
lock_request* request;                              /*this lock request block*/
TransCB* me=MyTransCB();                            /*pointer to caller's transaction descriptor*/
bucket= lockhash(name);                             /*find hash chain*/
Xsem_get(&lock_hash[bucket].Xsem);                  /*get semaphore on it*/
head = lock_hash[bucket].chain;                     /*traverse hash chain*/
while((head !=NULL) && (head- >name != name)        /*with this name*/
      {head = head- >chain;};
/*lock already taken in X mode : */
if ((head != NULL) && (head- >mode == X))
      { Xsem_get(&head- >Xsem);                     /*acquire semaphore on lock header*/
        Xsem_give(&head_hash[bucket].Xsem);         /*release semaphore on lock chain*/
        request = head- >queue                      /*the first request is the granted one*/
        if(request- >tran == me)
            return(LOCK_OK);                        /*ok to write an item already X-locked by the caller*/
        else if (request- >tran- >tn < me- >tn)     /*the read must be delayed ("critical read")*/
            { append(me- >pid,head- >pid_list);
              Xsem_give(&head- >Xsem);              /*release semaphore on lock head*/
              wait();                               /*wait*/
              return(LOCK_OK);
            }
        else
            return(LOCK_OK);
        }
}
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399